NPSCS-92-004

NAVAL POSTGRADUATE SCHOOL
Monterey, California

REAL-TIME SYSTEMS

Salah M. Badr, LTCOL, Egyptian Army
Ronald B. Byrnes, Jr, MAJ, USA
Donald P. Brutzman, LCDR, USN
Michael L. Nelson, MAJ, USAF

February 1992

Prepared for:

Naval Postgraduate School
Monterey, California 93943

NAVAL POSTGRADUATE SCHOOL

Monterey, California

REAR ADMIRAL R. W. WEST, JR.
Superintendent

HARRISON SHULL
Provost

This report was prepared for and funded by the Naval Postgraduate School.

Reproduction of all or part of this report is authorized.

This report was prepared by:

VALDIS BERZINS
Associate Chairman for
Technical Research

PAUL MARTO
Dean of Research

UNCLASSIFIED

RITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
DECLASSIFICATION/DOWNGRADING SCHEDULE	
ERFORMING ORGANIZATION REPORT NUMBER(S) NPSCS-92-004	5. MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School 6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
ADDRESS (City, State, and ZIP Code) Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943
NAME OF FUNDING/SPONSORING ORGANIZATION Naval Postgraduate School 8b. OFFICE SYMBOL (if applicable) NPS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER OM&N Direct Funding
ADDRESS (City, State, and ZIP Code) Monterey, CA 93943	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

TITLE (Include Secunty Classification)
Real-Time Systems

PERSONAL AUTHOR(S)
Salah M. Badr, Ronald B. Byrnes, Donald P. Brutzman, Michael L. Nelson

a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Summary	FROM _____ TO _____	1992 February 28	31

. SUPPLEMENTARY NOTATION

COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Autonomous underwater vehicle (AUV), hard real-time system, real-time operating system, real-time programming language, real-time system, soft real-time system

. ABSTRACT (Continue on reverse if necessary and identify by block number)

This paper presents an introduction to the basic issues involved in real-time systems. Both real-time operating systems and real-time programming languages are explored. Concurrent programming and process sychronization and communication are also discussed. The real-time requirements of the Naval Postgraduate School Autonomous Under Vehicle (AUV) are then examined.

DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
a. NAME OF RESPONSIBLE INDIVIDUAL Michael L. Nelson	22b. TELEPHONE (Include Area Code) (408) 646-2026 22c. OFFICE SYMBOL CS/Ne

FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Much has been written about the requirement of real-time control systems, whether these systems involve national defense, commercial production, or consumer products. Increasingly, everyday life is touched, either directly or indirectly, by real-time systems. But what are these real-time systems? Why are they needed, and what differentiates them from 'nonreal-time' systems? Are systems dubbed as real-time universally recognized as such? Is the function of a real-time controller nothing more than a robust scheduling mechanism with an impressive name? These questions and more will be touched upon herein.

The association of a controller and its 'critical' system has often been referred to as a *real-time system*. These systems are also sometimes referred to as *embedded systems* [BW90]. Real-time systems may be classified as either *hard* or *soft*. Although in wide-spread use, the definitions of these terms are far from consistent. Besides providing an introduction to the topic of real-time systems, the various meanings of these terms will also be explored.

The intended audience of this report is primarily graduate-level computer science students with a general background in programming languages, computer networking, and operating systems. Therefore, the basic concepts of these areas will not be discussed. However, the real-time aspects of each will be.

This report begins with an introduction to real-time systems and their terminology. Section 2 then discusses the issues related to real-time operating systems. Programming language and software engineering aspects of real-time systems are discussed in Section 3. The practical issues of real-time systems as they pertain to an example application (the NPS AUV) are explored in Section 4. Finally, our conclusions are presented in Section 5.

1.1 BASICS

The term real-time means different things to different people. Several definitions, some of which contain exquisite subtleties, are compared and contrasted here.

One way to define a real-time system is that it is "...characterized by the fact that severe consequences will result if logical as well as timing correctness properties of the system are not satisfied" [SR88, p.1]. Nuclear power plants, space shuttles, airplanes, and command and control systems are all clearly covered by this definition. However, what constitutes "severe consequences" and satisfaction of the "timing and logical" properties mentioned above? If your subway is late causing you to miss a job interview (and subsequently desired employment), is that "severe" enough to qualify the subway as a real-time system?

Most definitions of real-time systems found in scientific and engineering literature generally involve a relationship between a hardware/software control mechanism and a plant (a system whose operation is to be controlled), the function of which affects lives, property, or both. Recently, particularly in the area of consumer electronics, the real-time label has been applied to those systems whose operational time is equivalent to human time, even though their effect on our lives is minimal. The result of this trend is to broaden the class of systems at the expense of exact terminology, and to confuse unwary newcomers to the field.

What, then, is the implication of time in a real-time system? This simply refers to any timing constraints which are imposed on the system. That is, the output of the system is required to be not only computationally correct, but also temporally current (i.e., completed by a certain time). In systems of this type, correct results produced late may actually be worse than incorrect results produced before the deadline!

Another source of confusion is that real-time systems often interface with larger, non-real-time systems. A key feature of these applications is the role of the computer as an information processing component with a larger system [BW90]. The host system in which the real-time controller is embedded provides the data gathering and storage capabilities and other related resources to the computer. In addition, this larger system performs the work specified by the controller to alter the external environment. For this reason, real-time systems are often referred to as embedded systems and vice versa.

1.2 HARD REAL-TIME SYSTEMS

Real-time systems are often categorized as being either hard or soft. The generally accepted definition of hard real-time systems are those systems in which the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced [SR88, p.1]. The category of soft real-time systems includes all other real-time systems; these are not of foremost interest. Therefore, the term real-time will henceforth be used to mean hard real-time.

Of necessity, real-time systems are tailored specifically to their application, employ various degrees of fault-tolerance, and are typically embedded in larger systems. Features vary among such systems, but certain common characteristics do exist. For example, applications requiring real-time control are mathematically complex; hence, a substantial amount of raw information must be sampled, processed, and made available for use in control algorithms. As a direct result, the language in which the real-time controller is written must have the ability to manipulate floating-point numbers quickly and with a high degree of precision. Speed of execution is dictated by the timing constraints placed on the controller as well as the overall efficiency of the system. Two methods used to meet the required level of performance are program modification ('bit-diddling') and the use of multiprocessors.

In order to squeeze as much performance out of the code as possible, designers/programmers of real-time systems often make implementation decisions based on a specific target computer system. This tendency accounts for rigidity of existing systems and usually results in the need to start from scratch when designing a controller for a new application. The second method for enhancing performance is to utilize multiprocessors, although writing software that supports true concurrency is very difficult. This is due to a dearth of formal software engineering methodologies in this area, along with the lack of compiler support for parallel programming. Therefore, scheduling duties often fall upon the *real-time operating system*, sometimes called a *real-time kernel.*

Another common feature of real-time systems is the ability to function correctly despite unanticipated hardware and/or software failure, and/or human error. User confidence in the system can only be achieved if the system can be shown to be dependable and reliable. It may be the case, however, that these features may not be proven until true disaster looms. Nevertheless, solutions to such problems can be provided by the *fault-tolerant* capabilities of the real-time operating system or real-time kernel.

A *failure* in a real-time system can be defined as a deviation between the behavior of the real-time system and that which is specified. Failures result from either a mechanical (hardware) or algorithmic *fault* (software). Real-time systems are concerned with both the concepts of fault prevention and fault tolerance. *Fault prevention* is concerned with eliminating faults from the system before it goes on line. *Fault tolerance* is concerned with attempting to continued acceptable operation even if a fault does occur.

There are two stages to fault prevention: *fault avoidance* and *fault removal.* Fault avoidance attempts to limit the introduction of potentially faulty components during the design and implementation of the system. Despite the best efforts of avoidance techniques, however, faults will inevitably be present in the

system. It is then the concern of fault removal to find and remove the causes of these errors. This process is performed during system testing. For systems of even moderate size, it is usually impossible to perform exhaustive tests under realistic conditions. Therefore, designers of real-time systems must also rely on the concept of fault tolerance.

Fault tolerance itself involves a spectrum of techniques, including full fault tolerance, graceful degradation, and failsafe. A combination of techniques, called protective redundancy, is often used. Examples of redundancy include independent operational subcomponents with majority-voting circuitry and N-version programming [BW90]. Whatever technique is used for the achievement of fault tolerance will involve the introduction of extra elements into the system for detection, isolation, recovery, etc. It should be noted, however, that these elements add to the overall complexity of the system while reducing its efficiency. Execution time of the fault-tolerance component overhead must not cause timing constraint violations in an otherwise correct system.

Another difficulty faced by real-time systems occurs in situations where timing requirements either cannot be met or where they are dynamically changing. In these cases, it is again the operating system or kernel that must provide viable fixes. Although not a pure instance of fault tolerance, these types of problems also come under the province of fault tolerance.

Mean-time-between-failure (MTBF) is concerned with how long a system can be expected to successfully function before a problem arises. A software analogy to this concept is software reliability. This value can be considered as the probability that a given program will operate correctly in a specific environment for a given length of time. It also gives an indication of how well the system conforms to its specifications. Methods for finding this figure are not straightforward, however, and any published measure of reliability must be scrutinized with great care.

2 REAL-TIME OPERATING SYSTEMS

A real-time operating system provides many of the services typically associated with general operating systems. However, the user of such an OS is a real-time system; rarely are human users in the loop. For this reason, the real-time operating system is designed to perform a few functions very efficiently, functions such as resource management and job scheduling. Resources in real-time systems are intentionally kept to a minimum. Printers, terminals, and other human-interface devices are generally not dealt with because they are not required by the real-time controller. Scheduling, on the other hand, is of paramount concern to the real-time system. Should a scheduling conflict arise, the operating system is responsible for determining the order of task execution so as to maintain the integrity of the system.

2.1 CONCURRENT PROGRAMMING

Scheduling conflicts would not be a problem if the real-time controller were only concerned with a single sequential process. However, the requirement for rapid response, the vast amount of input data to be processed, and the significant fault tolerance efforts generally required preclude this type of implementation. Hence, the real-time controller, in order to satisfy the many constraints placed on it, must by necessity control the concurrent execution of several semi-independent (asynchronous) processes.

A concurrent program consists of a collection of asynchronous processes, executing (logically) in parallel. The implementation (i.e., execution) of such processes will take one of three forms: (1) multiplex their executions on a single processor; (2) multiplex their executions on a multiprocessor system where there

is a shared memory; or (3) multiplex their executions on several processors which have no shared memory. Only in cases (2) and (3) is there the possibility of true parallel execution.

The real-time operating system typically contains a scheduling algorithm referred to as the Run-Time Support System (RTSS) or Run-Time Kernel (RTK). This component manages the creation, dispatch, and termination of processes. In some cases, the RTSS may exist as a system generated by the compiler (as in Ada or microcoded into the host hardware (as in occam [PM87]).

2.2 SYNCHRONIZATION AND COMMUNICATION

The correct behavior of a concurrent program is critically dependent on the *synchronization* of concurrently executing processes. Synchronization involves the satisfaction of all constraints which arise when the execution of different processes are interleaved. These concepts and the difficulties relating to them will be discussed briefly in this section.

Synchronization is needed when one process wishes to perform an operation that can only be performed safely if some other process has already taken some action. For example, in the producer-consumer problem [Dei90, FM91], a buffer is shared between one or more producer processes and one or more consumer processes. Producer(s) must wait if the buffer is full, and consumer(s) must wait if the buffer is empty. Since the progress of the processes involved depends upon the existence of certain conditions, this requirement is more precisely called *condition synchronization.*

Closely related to process synchronization is the concept of *interprocess communication.* Communication is simply the passing of information from one process to another (in essence, condition synchronization is contentless communication). This exchange may take place using shared memory or via some form of message passing. Note that either form of communication can, in theory, be implemented to obtain the same functionality [SR88].

2.2.1 SHARED MEMORY SYSTEMS

Implementing data communications using shared variables has the disadvantage of being unreliable and unsafe due to multiple update problems. When a process is accessing the shared data, it is said to be in a *critical section* (CS). No other process may be accessing the shared data (i.e., in its own CS) during that time.[1] The synchronization required to protect a critical section is known as *mutual exclusion* (ME). It is the concept of mutual exclusion which lies at the heart of most concurrent process synchronization mechanisms [SR88].

The implementation of any form of synchronization implies that processes must, at times, be delayed from proceeding so as to ensure the proper order of execution. This is often discussed in terms of an entry protocol and an exit protocol which are placed around the critical section. The entry protocol prevents the process from entering the critical section when necessary, and is also responsible for indicating that this particular process has entered its critical section (and therefore no other process may enter its own critical section). The exit protocol indicates that the process has completed its critical section. Each process contains code of the basic form shown in Figure 1.

---

[1]It should be noted that it is possible to have several process reading the shared data at the same time, or there may be a single process writing the shared data (with no active reading processes) - this is the basic premise of the 'Readers and Writers' problem [Dei90, FM91].

```
process X;
   loop
      non-CS;
      entry_protocol;
         CS;
      exit_protocol;
      non-CS;
   end loop;
end X;
```

Figure 1 Using ME Protocols

Problems arising from various inadequate solutions to these algorithms (i.e., the implementation of the entry_protocol and the exit_protocol) include [Nel91]:

(1) ME violation, where competing processes are allowed to enter their respective CSs simultaneously;

(2) requests to enter the CS being denied even though no other process is in its own CS;

(3) deadlock, where competing processes are never allowed to enter their CS; and

(4) indefinite postponement (starvation), where some process(es) may never gain entry into their CS.

The entry and exit protocols may be implemented using a looping construct called busy waiting (BW). Busy waiting is a tight, polling-type loop in which some condition is continuously checked. When the proper condition exists, the loop is exited and the process proceeds. If the process then enters its critical section, additional safeguards may be involved to ensure that no other process may proceed with its own critical section. Implementations utilizing busy waiting are subject to indefinite postponement. This is due to the fact that if several processes are busy waiting on the same condition, there is no guarantee as to which one will proceed next - any process may simply never be lucky enough to finally get a turn.

One technique for synchronizing processes requiring ME (popular because it can be realized using software alone) is to have the process check and set various 'flag' variables. These flags are implemented using a combination of local and global variables. Well known examples of this technique include Peterson's algorithm and Dekker's algorithm [Dei90].

Semaphores [Dei90, FM91] are another software approach to the mutual exclusion problem. A semaphore is a flag (a signalling device), typically used to indicate when a resource is available. It can be thought of as an abstract data type with 2 operations, P and V.[2] The P operation is used to check to see if the resource is available, waiting if necessary; and the V operation is used to return a resource, signalling waiting processes if necessary. Semaphores have been implemented in many different ways, some utilizing busy waiting, and others utilizing some form of blocking (queueing). Regardless of the implementation, the P and V operations are atomic; that is, once execution has begun, they cannot be interrupted before completion. To utilize semaphores, each process now contains code of the form shown in Figure 2 (where mutex is a semaphore established to maintain mutual exclusion for some shared resource).

Various special hardware instructions have also been developed. These include 'swap', 'test and set', and 'test, set, and branch' [Dei90, FM91]. As hardware instructions, they have the advantage that they are guaranteed to run atomically. Unfortunately, entry and exit protocols that rely solely on these special hardware instructions generally suffer from busy waiting [Nel91]. However, combined hardware/software

[2]Unfortunately, P and V do not carry much meaning in the English language. P stands for proberen (to test) and V stands for verhogen (to increment) [FM91].

solutions may be implemented which take advantage of the specialized instructions, using software to implement a waiting queue [Nel91].

```
process P;
   loop
      non-CS;
      P(mutex);
         CS;
      exit_protocol;
      V(mutex);
   end loop;
end P;
```

Figure 2 Semaphore Implementation of ME

Another software structure called the monitor [Dei90] can also be used to enforce mutual exclusion. A monitor contains both data and procedures which are used to control the sharing of resources. Only one process may be executing within a monitor at any time. A process requests to make an *entry* into the monitor when it is requesting or returning a resource. When requesting a resource that is unavailable, the process is blocked (i.e., placed on a queue which is called a condition within the monitor). When a resource is returned, the first blocked processes waiting on that resource is then signalled (notice that signalling an empty queue has no affect). Figure 3 contains a simple monitor, and Figure 4 shows how a process requests its services.

```
monitor resource_monitor
var  resource_busy: boolean;
     waiting_queue: condition;
procedure get_resource;
  begin
    if resource_busy
      then wait(waiting_queue);
    resource_busy := true;
  end;
procedure return_resource;
  begin
    resource_busy := false;
    signal(waiting_queue);
  end;
initially
  begin
    resource_busy := false;
  end;
end monitor;
```

Figure 3 A Simple Resource Monitor

```
process X;
  loop
    non-CS;
    resource_monitor.get_resource;
      CS;
    exit_protocol;
    resource_monitor.return_resource;
  end loop;
end X;
```

Figure 4 Using the Resource Monitor

The rendezvous in Ada can also be used to manage the sharing of resources [Bar89, Boo87, Dei90]. A calling routine calls an entry in a serving routine which accepts that entry when it is ready to do so (thus, we have a rendezvous between the caller and the server). A simple resource controller in Ada is very similar to the concept of the monitor, except that it is the Ada environment which establishes the waiting queues. Figure 5 gives an example of a simple resource controller. Our process X now simply calls the routines 'resource_controller.get_resource' and 'resource_controller.return_resource'.

```
task resource_controller is
  entry get_resource;
  entry return_resource;
end resource_controller;

task body resource_controller is
begin
  loop
    accept get_resource;
    accept return_resource;
  end loop;
end resource_controller;
```

Figure 5 Ada's Rendezvous

Unfortunately, all of these techniques are subject to various weaknesses and problems. As previously discussed, any technique employing busy waiting can lead to indefinite postponement. These systems do not guarantee that a process cannot erroneously (or maliciously) return a resource that it does not have (although a programmer may be able to implement this).

2.2.2 MESSAGE PASSING AND DATA COMMUNICATION

There are three broad classifications of process synchronization, all based on the semantics of the send operation: *asynchronous*, *synchronous*, and *remote procedure call* (also called a remote invocation). In a system using asynchronous communication, the sender continues processing after sending the message, regardless of whether or not it was actually received. With synchronous communications, the sender proceeds only after the message has been received (this is the form found in occam2). A remote procedure call does not allow the sender to proceed until the message has been received, read, and a reply is received from the receiver (this is also called the request-response paradigm, and is found in Ada).

Asynchronous message passing is sometimes called a nonblocking send, in that the sender is not blocked in any way. This is illustrated in Figure 6.[3]

Synchronous message passing is sometimes called a blocking send, in that the sender is blocked until the message is received. This is illustrated in Figure 7.

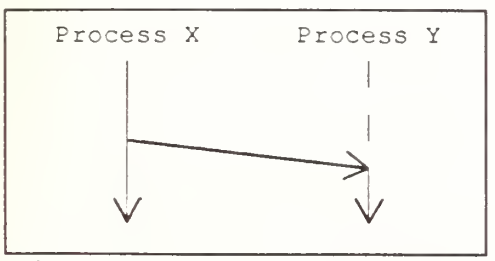

Figure 6 Asynchronous Message Passing

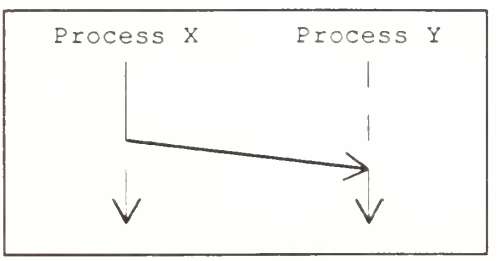

Figure 7 Synchronous Message Passing

[3]Message passing figures are adapted from [TS89].

A remote procedure call (also called a remote invocation) is another form of synchronous message passing, except that the sender is blocked until a reply is received. This is illustrated in Figure 8. A variation on the remote procedure call is the future remote procedure call. Now, the calling process may continue after sending a message, but only until the reply is needed for it to continue (note that if the reply is received before the deadline, then the calling process is not blocked at all). This is illustrated in Figure 9.

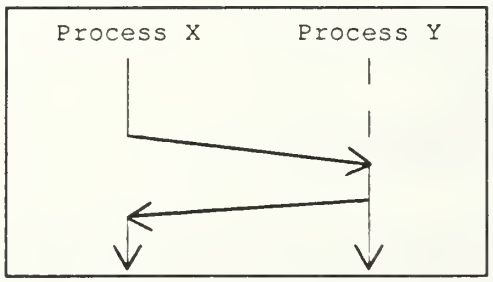

Figure 8 Remote Procedure Call

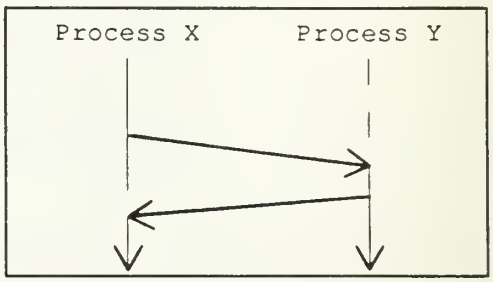

Figure 9 Future Remote Procedure Call

To account for all situations arising in real-time applications, concurrent programming languages must allow a process to choose between alternative communication models. This is often referred to as *selective waiting* [BW90]. In Ada, for example, selective waiting is provided by the 'select' statement.

2.3 RESOURCE CONTROL

Many of the issues discussed in this section are closely linked to the language used in implementing the real-time system. Resource management may be controlled either by the implementation language or by the operating system. As systems grow in size and complexity, however, resource control is better dealt with by the operating system, since it is a more central entity.

Processes which require a resource normally request the resource, use it, and then release it upon completion. Resources may be requested for shared access (as in read-only files) or exclusive access (as for CPU time). Depending upon the mode used, a requesting process may be required to wait. Since a process waiting for a resource may be blocked, it is incumbent upon the operating system to insure that resources are not scheduled until needed and that they are released immediately after the using process has finished with them.

With many processes competing for nonshared resources, the problem of deadlock becomes very real. Two processes are said to be deadlocked if they each have exclusive access to a resource that the other is requesting. Another problem related to the management of resources is that of indefinite postponement (also referred to as starvation or lockout). If the allocation policy used by the scheduler is not 'fair', then some process(es) among several that are competing for a same resource may never gain access to that resource. The unlucky process is starved of the resource and will subsequently be blocked indefinitely.

Four conditions must hold for deadlock to occur:

(1) mutual exclusion: the resource is nonsharable;

(2) hold and wait: existing processes are holding some resources while waiting on others;

(3) no pre-emption: only the process holding a resource may release it; and

(4) circular wait: a circular chain of processes must exist such that each process holds a resource that is being requested by the next process in the chain.

Real-time systems must address the issue of deadlock if they are to be considered reliable. Deadlock strategies are typically concerned with either prevention, avoidance, or detection (and recovery). The reader may refer to [Dei90] for more information on these strategies.

2.4 DEADLINE SCHEDULING

The notion of time as it relates to real-time programming languages may be described in terms of the following four requirements: access to a real-time clock; the ability to delay; the ability to limit waiting via timeouts; and the specification of deadlines and scheduling. Interestingly, even though deadline scheduling is what differentiates real-time programming from conventional programming, facilities for this type of scheduling are, for the most part, inadequate in the recognized real-time languages.

2.4.1 TELLING THE TIME

Simply put, real-time programs must organize and meet execution deadlines on the host's time. This can be done by utilizing available clock primitives provided by the language or by creating the necessary drivers to access the system's clock. An example of the former is Ada, with its library package CALENDAR [DOD83].

2.4.2 DELAYING A PROCESS

A process may need to delay its execution until some point in the future. Rather than relying on a busy wait loop which continuously polls the clock, the process must be able to delay itself a specified amount of time. Ada again provides an example with its 'delay x' statement, whereby the process will be delayed (at least) x seconds from the time that this command is encountered. Notice, however, that the delay may last longer than x seconds - the process is simply placed in the ready-to-run queue x seconds later; if the CPU is busy at that time, then the process will have to wait even longer than x seconds.

2.4.3 PROGRAMMABLE TIMEOUTS

More manageable is the circumstance where an event does not occur. These constraints are the simplest for real-time systems to identify and process. In general, the maximum length of time a suspended process will wait for an external event is embodied in the timeout. No special purpose statements for timeouts are provided by Ada, although the required functionality is available using the 'select' construct [BS86].

2.4.4 DEADLINE SPECIFICATION AND SCHEDULING

To satisfy the deadlines (timing constraints) of real, physical systems, simple timeouts are not sufficient. For this reason, these constraints must be incorporated into the concurrency model of the real-time language. The constraints, in turn, become visible characteristics of the software processes. Such processes can be classified into periodic and aperiodic tasks. Periodic tasks are driven by explicit deadlines, while aperiodic tasks arise from asynchronous external events. Regardless, all processes must be analyzed to determine their worst-case (or average) execution time.

Specification of timing constraints typically involves the concepts of temporal scope, hardness, and priority. Temporal scope (TS) identifies a section of the application program with a deadline. Possible attributes of a temporal scope include the deadline, a minimum and maximum delay associated with the start of execution of the TS, the maximum execution time of the TS, and the maximum elapse time of the

TS. Given such information, the problem of satisfying timing constraints becomes one of scheduling to meet these deadlines, otherwise known as deadline scheduling.

Process scheduling within a real-time system is often provided by the run-time dispatcher using a priority scheme. In soft real-time systems, the use of priorities and a preemptive scheduling algorithm will usually suffice. The priority indicates which processes are to be executed in situations where some deadlines cannot be met. In hard real-time systems, of course, all deadlines must be met, so this preference priority scheme is of less value.

The lack of explicit deadline scheduling control in Ada requires that the run-time scheduler assume the responsibility of deadline maintenance. Some real-time languages which do provide such support include Real-Time Euclid [KS86], PEARL [WW85], and DPS [LG85]. When using these languages, the run-time scheduler is much easier to implement; in fact, such a scheduler may even utilize a static-priority scheme.

Once the deadlines associated with each process in a real-time system have been determined, a schedule compatible with these deadlines must be derived. A general algorithm which checks for the existence of such a schedule doesn't exist; instead, several schemes can test for schedulability under certain conditions. Typically, a fixed number of processes are analyzed according to their individual execution times, inter-process communication timeout values, and measure of utilization. Language constructs such as dynamic memory allocation, dynamic creation of processes, and recursion are often not allowed in real-time processes in order to facilitate the predictability of these parameters. Unfortunately, no rules-of-thumb for scheduling hard real-time processes exist with the exception that all such processes must be scheduled using worst-case execution times. Of course, one drawback of the resulting schedule may be an unacceptably low utilization of resources.

Once the schedule is set, the real-time scheduler must assume the responsibility of seeing that the schedule is followed. The purpose of the scheduling algorithm is to ensure that the processes are dispatched in an order which meets all deadlines. That is, the algorithm controls the behavior of the dispatch queue. Again, no general panacea exists, although research has produced schedulers based on many different priority schemes. Examples include period-priority, earliest deadline, and least slack time as bases for scheduling [BW90].

3 LANGUAGES FOR REAL-TIME SYSTEMS

The timing, execution, and resource constraints of real-time systems dictates the following requirements for any language used in building them [SR88]:

- Modules should have predetermined bounded execution times.
- The use of dynamic structures such as dynamic arrays, pointers, and arbitrarily long strings should be controlled.
- Execution time of all modules must be known at the time a module is scheduled (thus recursion and loops must be used very carefully).
- Dynamic process creation should be restricted to permit a priori determination of resource needs of processes in the system.
- Provision should be made for all known types of exceptions to guarantee completion of each module, and for the system behavior to be predictable.

- All program constructs and modules should be analyzable with respect to schedulability of modules, availability of resources, and meeting of timing constraints of the system in order for the system to be predictable.

3.1 APPROACHES TO PROGRAMMING REAL-TIME SYSTEMS

There are four basic approaches to programming real-time systems. They all share a common characteristic though, and that is to minimize the execution time to make the program meet its timeliness. The four basic approaches are as follows [NL88]:

- Using assembler or other low level language. This approach, besides being difficult, is not easily reusable, modifiable, or portable, and it is very hard to test.
- Using languages such as Ada or Modula_2. Knowledge of the run time environment is required to tailor the program to meet its timing specifications which make the program sensitive to hardware characteristics and system configuration. Additionally, these languages do not permit explicit expression of timing requirements.
- Restricting the constructs provided by the language to those which are time-bounded. The program may not use recursion, dynamic memory allocation, or dynamic process instantiation because its execution time would then be unpredictable. This approach facilitates estimation of execution time of the program, but makes writing programs more difficult using only a restricted set of constructs.
- Direct expression of timing requirements. The program specifies the deadlines for procedures, and provides for exception handling if the procedure is terminated prematurely. The run-time system ensures that the specifications are met.

3.2 FLEX: A REAL-TIME PROGRAMMING LANGUAGE

As one example of a real-time system language, we will now discuss FLEX [NL88, SK88] with special emphasis on timing issues and the flexible execution that it offers so that programs meet their deadlines. FLEX presents new concepts which can be used with existing functional, object-oriented or procedural languages [NL88]. The current implementation is a modification of the object-oriented language C++.

FLEX programs consist of a set of objects. Each object consists of statics (which store values) and procedures (which access and modify the statics). The statics store state information and the procedures cause state transitions. The only form of communication between objects is to have one procedure invoke another procedure, either in the same object or in a different one. Procedures may also invoke functions to perform computations.

Statics are declared inside the object and keep their values across procedure calls. The variables local to procedures and functions are called locals. Locals may be declared inside the body of a procedure or a function, and are known only within that procedure or function. They are re-initialized each time the procedure or the function is called. Statics and locals are both referred to as items.

3.2.1 THE CLOCK OBJECT

FLEX defines a clock object for modeling the notion of time, which includes the following procedures [NL88]:

get-time: returns a structure containing the current local time (including the date, hour, minute, second, and fraction of a second).

delay: returns when the specified time interval expires.

tick: an internal procedure which updates the clock periodically.

The clock object enables the programmer to indicate relative or absolute times. For example, you can specify that a response must be received within n seconds from the time the input is received (i.e., the deadline is relative to the current time), or by a specific clock time (i.e., the deadline is absolute).

3.2.2 TIMING CONSTRAINTS

In FLEX, constraints are used to express a relationship among the items related to these constraints. The relationship must be established and maintained throughout the scope of the constraint. Exception handling may be specified (as an option) when the constraint is violated. The scope of a constraint is defined as "the statement or the group of statements, called a constraint-block (CB), that follow it" [NL88, p.274].

For timing constraints to be expressed, each CB has the following features defined:

(1) start: the absolute time at which execution of the CB begins.

(2) finish: the absolute time at which execution of the CB finishes.

(3) duration: the time interval during which the CB executes; it is the time interval from start to finish, except for periodic CBs in which case it is the execution time of a single iteration of the CB.

(4) period: the time interval between successive executions of a periodic CB.

(5) priority: an integer value indicating the importance of the CB (default priority is zero).

A timing constraint is a boolean expression involving any of these features combined with items using arithmetic and logical operators. A periodic CB is one for which the period feature is specified. It is executed repeatedly every period until the constraint becomes false. For periodic CBs the start, finish, and priority features indicate the entire execution of all the loops. The exception handler is called only if the duration or period features are violated for a specific iteration of the CB.

To show the features of FLEX, consider the example of a robot moving along a straight path.[4] There may be other objects blocking the path or moving along it, so the robot has to adjust its speed as necessary or move aside to avoid slower or stationary objects (assuming that the path is wide enough). The robot eyes provide it with vision signals and distance estimates. An algorithm to control the robot could be as follows:

```
loop every one second
    {Interpret vision signals and check for hazardous conditions;
     If no hazards, resolve positions of all surrounding objects;
     Decide and adjust the speed and direction of movement;
    }
```

[4]This example is taken from [NL88].

An example of a CB could then be as follows:

```
Z1: danger = false -> react(object);
{for each object do
    compute_distance(object);
}
```

The CB is defined by the loop. It tests the safety of the robot with respect to other objects. Z1 is the label for the CB. If the object is close enough, compute_distance sets the danger flag to true, which indicates the constraint violation, and the exception handler 'react' is invoked (the symbol '->' denotes exception handling).

The CB could also be a null statement. This constitutes a timing constraint to be tested, and an exception is raised if a constraint violation is detected. For example, the following constraint can be used to indicate that the deadline is close:

'Clock.get_time < (loop_start + 0.7) -> quit_loop := true; {}'

As an example of a timing constraint, consider the procedure of deciding whether to move aside. Since the entire loop must complete within 1 second, the decision-making procedure must start in time and finish while there is still time left to carry out the decision; assigning a higher priority will help to ensure that this occurs. The following constraint can be used at the proper place inside the loop:

Z2: (start < loop_start+0.7) && (finish < loop_start+0.8) && (priority = 8)

loop_start is set at the start of the loop by a call to Clock.get_time(). The loop itself is a periodic task which must be completed every second. If it cannot be finished in time, then we would like to implement the decision reached so far. Also, if a hazardous condition is detected we want to respond immediately to it rather than wait until the analysis is done. So we can use:

Z3: (period = 1) && (!danger) -> take_action(decision);

Vision images are normally sharpened (i.e., edge enhancement is performed) before attempting to recognize objects in them. This is not crucial, hence we may set an upper limit of 10% of the loop time to be spent on this process by using the following:

'Z4: (duration < 0.2 * Z3.period) && (priority = -4)'

Thus, we see that a constraint may refer to the timing attribute of another CB.

3.2.3 OTHER FEATURES OF FLEX

FLEX introduces the notion of imprecision, which allows the starting of any calculation by first computing approximate results and then, as time permits, improving these results by using more precise ones. This means if a time constraint is violated we have the flexibility to implement the decision reached so far (using approximate results) instead of calling an exception handler which, in general, cannot produce meaningful results. This feature is the main reason for naming this language FLEX. [NL88]

3.3 REAL-TIME FEATURES OF ADA

As a language for real-time applications, Ada includes a package CALENDAR that exports a TIME type and a predefined function CLOCK that returns the time of day. It also includes the package STANDARD, which includes a predefined DURATION type, that is used to indicate units of seconds. The delay state-

ment in Ada takes one argument of type DURATION and suspends further execution of the job for at least the given time interval (after the duration of the delay is expired the job is returned to the ready queue to wait its turn for execution according to its priority). The argument for the delay statement may be a constant or simple expression of type DURATION. For example:

```
delay 20.0                              -- delay 20 seconds

delay NEXT_TIME - CALENDAR.CLOCK        -- delay for difference between NEXT_TIME
                                           and time of day (CALENDAR.CLOCK is a
                                           function call that returns the time of day)
```

Ada does not provide attributes to explicitly express timing constraints to define the start, finish, and duration of a constrained block. The Ada9x committee [Ada90] is considering additional real-time features, so we will most likely see at least some of these features in future versions. But until then a knowledge of the run-time environment is a must to tailor the program to meet its timing specifications. Unfortunately, this makes the program sensitive to hardware characteristics and system configuration.

4 NPS AUTONOMOUS UNDERWATER VEHICLE (AUV)

The Naval Postgraduate School (NPS) Autonomous Underwater Vehicle (AUV) is an eight foot, 387pound untethered robot submarine designed for research in adaptive control, mission planning, mission execution, and post-mission data analysis [HMC90]. The NPS AUV is typical of other autonomous robots in that a large number of processes must run simultaneously while meeting stringent real-time requirements. The NPS AUV differs from other robots in that it is designed to operate submerged and isolated from communication or external directions, thus requiring extraordinarily reliable and robust vehicle performance.

Although numerous software modules have been written with the NPS AUV in mind, very little software has actually been implemented, integrated and tested underwater in real time. The main reason for this deficiency is the current lack of a flexible high-level software control module that can efficiently coordinate multiple AUV processes using the OS-9 real-time operating system.

Pertinent real-time operating system considerations for the NPS AUV are listed in Figure 10. These issues are examined in this section.

4.1 REAL-TIME REQUIREMENTS

In order to perform numerous sophisticated mission functions, multiple processes must be operating simultaneously while meeting both strict and relaxed real-time schedule requirements. The autonomous nature of an AUV requires operation without external backup in a harsh and unforgiving environment. Vehicle control, sensor evaluation, underwater navigation, search, path planning, obstacle avoidance, fault tolerance, and numerous other processes are required. All processes must interact with the external environment and each other in real time with varying degrees of interdependence [Bob91].

It is important to distinguish between hard and soft scheduling criteria for real-time processes. Mission-critical actions such as vehicle control and failure detection are hard real-time scheduling requirements. Failure to meet such hard deadlines may result in mission failure or even catastrophic loss of the vehicle. Conversely, high level logical processes such as path planning or mission replanning might always be considered soft requirements, since their execution is rarely mandatory for safe vehicle operation and immediate results are not required. Finally, some processes have priorities that may vary from soft to hard

- NPS AUV and Real-Time Operations
- Hard and Soft Real-Time Requirements
  - Multiple processes concurrently operating without backup in a harsh and unforgiving environment
  - Vehicle control and mission critical actions are hard
  - High-level logic function priorities may vary from soft to hard depending on circumstances
  - Numerous soft requirement processes exist, may compete independently or be mutually dependent
- AUV Process Deadline Specification and Scheduling
  - Simplistic mission software currently runs in single loop under a 10 Hz clock, which is not feasible for AUV software of even moderately greater complexity
  - Periodic and aperiodic tasks shown by data flow diagram
  - Rate monotonic scheduling thesis research at NPS; rate-monotonic scheduling under OS-9 can provide processor utilization above 80% and graceful degradation under overload
  - Dynamic scheduling
  - Modifications and new code constantly under development, require integration into system schedule
- Parallel Processing and Concurrent Programming
  - Mission requirements lead to multiple software modules operating independently and concurrently
  - Multiprocessing should be expected in order to achieve mission success
  - Some shared memory will be essential
  - Synchronization and communication methods available to processes are critical considerations, especially for process communication with the mission executor module
  - Extensions for distributed processing
  - Massively parallel processing is not a critical requirement
- Operating System Compatibility and Interoperability
  - Hardware: several different processors internal to vehicle; numerous analog/digital interfaces
  - Software: interfacing with different operating systems on each processor; multiple programming language support; process encapsulation to prevent faults and side effects; accessibility to individual machine-dependent, machine-level and device-dependent instructions and routines
  - External: mission data collection, consolidation, storage, and transmission; interaction with integrated simulator; network communications; distributed processing; communication with non-native operating systems and software environments
  - Other considerations: real-time implementations under Unix; common operating systems modified to handle real-time; TRON; distributed operating systems
- OS-9 Operating System
  - Hardware characteristics: 68020/68030/68040, GESPAC, VME bus, 80386, INMOS transputers
  - OS-9 process states: start (fork), active, run, exit, sleep, wait (process synchronization), and event wait (semaphore communication)
  - Features supporting expected AUV requirements: adjustable priorities and aging for execution scheduling, preemptive process switching, interrupts, traps, events, signals, pipes, redirection
  - Deficiencies and expected shortcomings: currently no Ada or C++, no deadlock protection, modifiable ROM system kernel pros/cons
- Current Problem Areas and Future Research
  - Deadlock prevention: expensive but essential; real-time deadlock detection may be a straightforward solution
  - Software engineering considerations: systems integration, verification, and validation, software version control, upward compatibility for future software modules
  - Fault tolerance: software checks vice redundant processors
  - Need formal specifications, characteristics, and timing constraints for all software modules in DFD
  - Establish new baseline architecture of system software running in the vehicle to allow more sophisticated operations and addition of new processes to basic control loop
  - Ensure maximum processor utilization through improvements to rate-monotonic scheduling algorithm, and evaluate incorporation of dynamic scheduling features
  - Investigate alternate system software architecture organizations, including multiple intelligent agents, low-level behaviors, expert systems, and blackboard paradigms; flexible support of multiple paradigms will allow greatest research progress

Figure 10 NPS AUV Real-Time Operating System Considerations

depending on circumstances. For example, obstacle avoidance is typically a soft requirement until target proximity or rapidly closing range rate make immediate action necessary to avoid collision.

4.2 PROCESS DEADLINE SPECIFICATION AND SCHEDULING

The current NPS AUV mission software schedule runs a single rudimentary control loop using a 10 Hz clock. A full 100 millisecond interval is allotted for each loop, but no processes are allowed to exceed that period. This interval is adequate to perform numerous tasks: compute basic vehicle control orders, transmit using one to four sonar transducers, record all current vehicle data parameters in working memory, perform rudimentary sonar analysis, detect waypoints, detect potential collision, and order predetermined state changes in propeller speed and control surface position. Typically very little time remains at the end of each fixed 100 millisecond time segment. Such a simple hard-wired timing mechanism is not a feasible control architecture for AUV software of even slightly greater complexity.

Proposed NPS AUV software modules are shown in the dataflow diagram (DFD) of Figure 11. Only the basic dependencies of these NPS AUV task modules have been characterized. A formal analysis of software module specifications, timing requirements, task periodicities, and concurrency dependencies has yet to be completed. It is likely that the NPS AUV software modules will ultimately have timing constraints and periodicity characteristics similar to those shown in Table I.

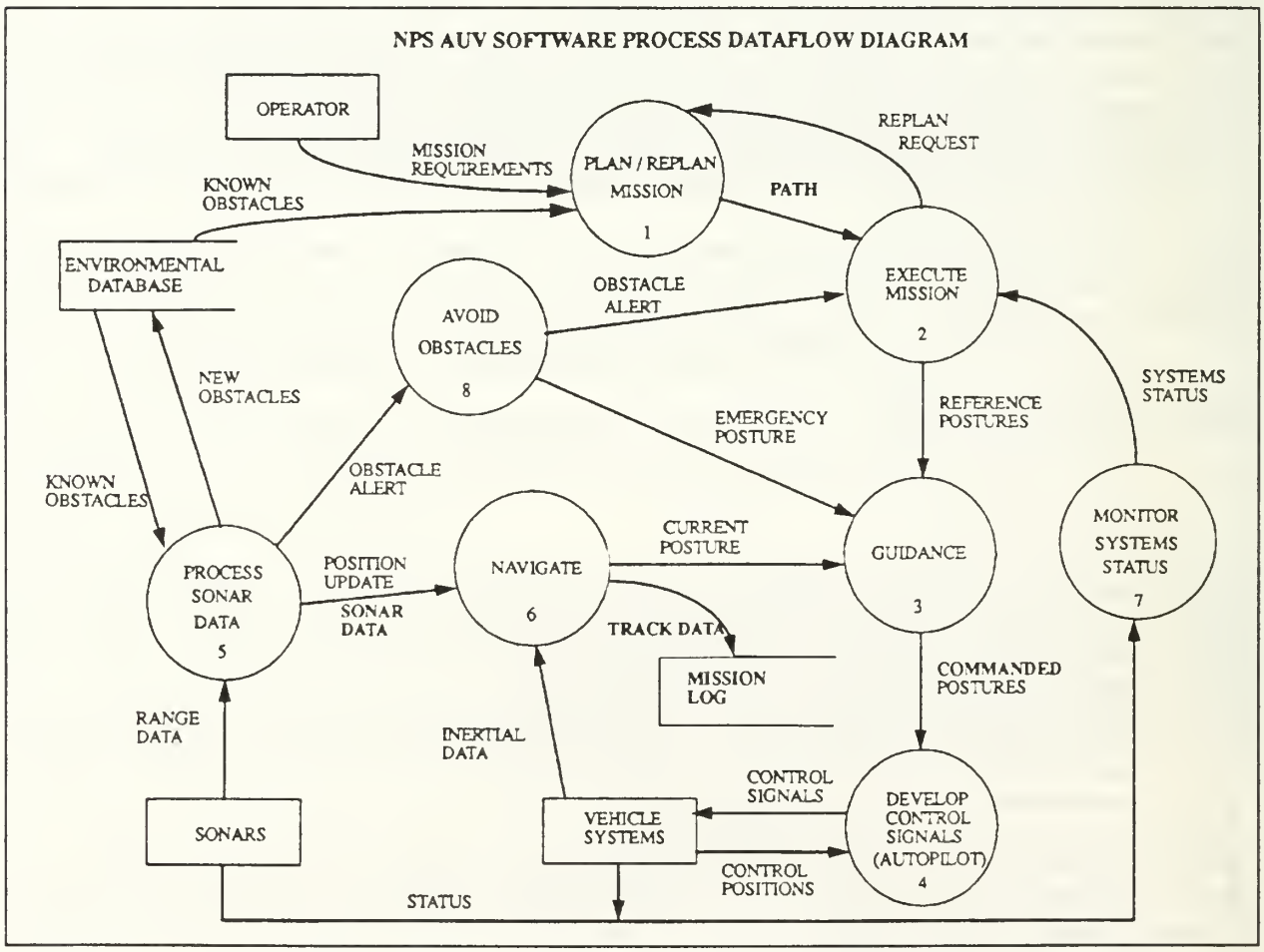

Figure 11 NPS AUV Software Process Dataflow Diagram
(Courtesy of Charles A. Floyd, CDR, USN)

AUV Software Modules	Real-Time Execution Characteristics		
Plan/Replan Mission	Soft	Aperiodic	Event triggered
Execute Mission	Hard	Periodic	Every control loop
Guidance	Both	Periodic	Every loop unless preempted
Autopilot	Hard	Periodic	Every control loop
Process Sonar Data	Hard	Periodic	Every loop unless preempted
Navigate	Soft	Aperiodic	Every loop unless preempted
Monitor Systems Status	Both	Periodic	Every loop unless preempted
Avoid Obstacles	Hard	Periodic	Every control loop

Table I NPS AUV Software Module Real-Time Characteristics

Current real-time operating system research at NPS has focused on rate-monotonic scheduling theory, an approach that employs fixed prioritization of processes and guarantees acceptable average performance [Lea91, Mak91]. Benefits of rate-monotonic scheduling include guaranteed completion of periodic tasks in order of priority, fast response for aperiodic tasks, modifiable task priorities, and the scheduling of tasks that permit imprecise computations (i.e. output precision proportional to time available). Rate-monotonic task analysis and scheduling is performed off-line prior to actual execution of system software. Rate-monotonic scheduling implementations running dummy processes under the OS-9 operating system have been shown to provide processor utilization above 80% and graceful degradation under overload.

Dynamic adjustment of constraints and process schedules may ultimately be required to ensure successful AUV operation during unforeseen tactical scenarios or pathological process conflicts. Dynamic scheduling theory requires further formal research to provide a verifiable theoretical foundation, but it appears to be a desirable model for the distributed artificial intelligence applications likely to make up the NPS AUV. In this regard the FLEX programming language is worth consideration since it implements dynamic scheduling theory and generates C++ code as output [KL91]. Additionally, there exists a hybrid approach known as a mixed priority system that combines the best features of rate-monotonic scheduling and dynamic scheduling [Lea91]. Formal evaluation of the mixed priority approach also appears worthwhile.

Given that AUV-related research is likely to continue for many years by several academic departments at NPS, operational software changes and additional new software processes will always be under development and require integration into the overall NPS AUV system process schedule. Reliability, compatibility and extendibility for future growth must be key attributes for any proposed control process timing schedule. Robust and flexible interactions between numerous interdependent processes will be essential to allow frequent improvements to vehicle performance while maintaining vehicle reliability.

4.3 PARALLEL PROCESSING AND CONCURRENT PROGRAMMING

It is important to note that parallelism is equally as important as real-time scheduling for an AUV operating system. This is particularly true if low-level control, complex behaviors, sensor fusion, data analysis, mission planning and numerous other artificial intelligence aspects of robot mission execution must all coexist and cooperate in a rapid manner [SR88].

Non-trivial robot performance requires that numerous processes operate in parallel, either independently or in a mutually dependent fashion [Kas88]. Numerous challenging AUV mission requirements will inevitably lead to multiple software modules operating concurrently. Such parallelism might be most easily implemented using a multiprocessor architecture. An AUV's real-time operating system must completely support concurrency constructs which are fully integrated with the real-time scheduling mechanisms.

Several standard features of parallel programming are necessary for effective software engineering of a real-time AUV. Adequate shared memory is essential if numerous processes are to quickly and efficiently access system state variables and the large amounts of time-sensitive sensor data that is expected. Predictable rendezvous, synchronization and communication methods must be available, both for interaction between mutually dependent processes as well as loose overall control by a mission executor module.

Increasing hardware sophistication can further allow tasks to be distributed over a network among separate specialized embedded processors [SR88]. Extensions of process rendezvous, synchronization and communication should be available for distributed processing if networked processors are to be employed.

Massively parallel processing in the classic sense uses numerous processors in parallel to perform array processing or numerous parallel solutions of identical algorithms. Such an approach is not a likely requirement for AUV operation. Aside from potential analysis of sophisticated sensor data using vision processing techniques, few (if any) AUV functions can be decomposed into numerous identical subproblems. The diverse nature of the many AUV software modules implies that a transputer architecture is not a prerequisite for successful integration of multiple AUV processes. Nevertheless, the transputer paradigm may be an effective way to minimize interface difficulties while allowing unlimited addition of numerous unique parallel processes under a single integrated real-time operating system. This approach is also being considered by C.S. Draper Laboratories for the next-generation software architecture of the DARPA Unmanned Underwater Vehicle (UUV) [Hal91].

4.4 OPERATING SYSTEM CONSIDERATIONS

It is important that an AUV operating system be fully compatible with all current and projected vehicle hardware and software components. External connectivity of the real-time operating system is also important.

Hardware interoperability considerations must consider connections between multiple processors of various types internal to the vehicle, as well as numerous analog/digital and digital/analog interfaces. Space, weight and power requirements are very strict so internal AUV hardware architectures must be closely compatible. Physical compatibility improves vehicle endurance by reducing power consumption.

Software compatibility is less critical than hardware compatibility, but software incompatibilities can still impose undesirable processing delays if too much work is required to translate communications between processors. Network support and software interfacing will be needed when different operating systems reside on multiple processors. Multiple programming language support is desirable for unrestricted research in a variety of control systems and artificial intelligence subjects. Process encapsulation is desirable in order to minimize faults and side effects during software development. High-level software access to machine-dependent, machine-level and device-dependent routines is also needed. Such routines permit various processes to utilize the operating system for direct access and control of the numerous physical components of the AUV.

Although the NPS AUV is untethered and isolated during operation, a number of external compatibility requirements remain. Mission data collection, consolidation, storage and transmission are ultimately targeted for external off-line post-processing and analysis. Distributed processing over a network internal to the AUV would require that each individual operating system must be able to interact with the others. Interactive network communication is also a likely requirement for on-line laboratory testing of the AUV. Connecting the vehicle or similar lab prototypes to the NPS AUV Integrated Simulator will allow scientific visualization of NPS AUV processes for active real-time end-to-end developmental testing [Bru91]. For these reasons the NPS AUV must be able to communicate with non-native operating systems and software environments. External connectivity is essential to support the diverse and distributed academic community that is conducting NPS AUV research.

Several other operating system considerations are worth noting. Real-time constructs and compatibility can be incorporated into typically non-real-time operating systems (such as Unix) by adding specially designed message-passing processes [Cra88, Fal88, Hil88]. Modified real-time kernels of common operating systems (such as Mach for Unix or FlexOs for DOS) are commercially available. As robotic systems and intelligent machines become more commonplace, the interactive design concepts of TRON (The Real-time Operating system Nucleus) will become increasingly important [Kah91]. Finally, a distributed operating system may provide the most efficient control mechanisms for distributed processors sharing distributed resources [DLA91].

4.4.1 THE OS-9 OPERATING SYSTEM

OS-9 is Microware System Corporation's real-time operating system used by the NPS AUV. OS-9 is designed to run exclusively on the Motorola 68020/68030/68040 family of microprocessors [GES89]. The two specific hardware configurations used in the NPS AUV include GESPAC 68020 or 68030 microprocessors connected to a Gesbus (VME bus compatible) backplane. Also available for immediate use is an Intel 80386 microprocessor although this processor has not yet been networked. Serial ports, parallel ports, analog/digital interface cards and an Ethernet interface are available for internal and external connections.

The OS-9 process states available include start (fork), active, run, exit, sleep, wait (process synchronization), and event wait (semaphore communication). Process state transitions are shown in Figure 12 [GES89].

OS-9 features that support expected NPS AUV operating system requirements include adjustable priorities and aging for explicit execution scheduling, preemptive process switching based on priority, reprogrammable interrupts, a trap library, events for process synchronization, signal communication between processes, pipes for interprocess data transfer, and redirection of process inputs and outputs [Dib88]. Identical syntax when referring to processes or device drivers is a particularly convenient feature of OS-9.

Deficiencies and expected shortcomings of OS-9 include the current lack of compatible Ada or C++ compilers and no simple method of deadlock protection. Additionally a fully modifiable operating system kernel must be prepared through careful EEPROM configuration prior to operation. This preparation allows setting up the operating system to include only the device drivers that are necessary for the current vehicle hardware. OS-9 is very flexible in that additional drivers may be loaded at any time after system initialization by software command. However, the EEPROM configuration process is time consuming, version dependent and error prone due to limited self-diagnostic testing.

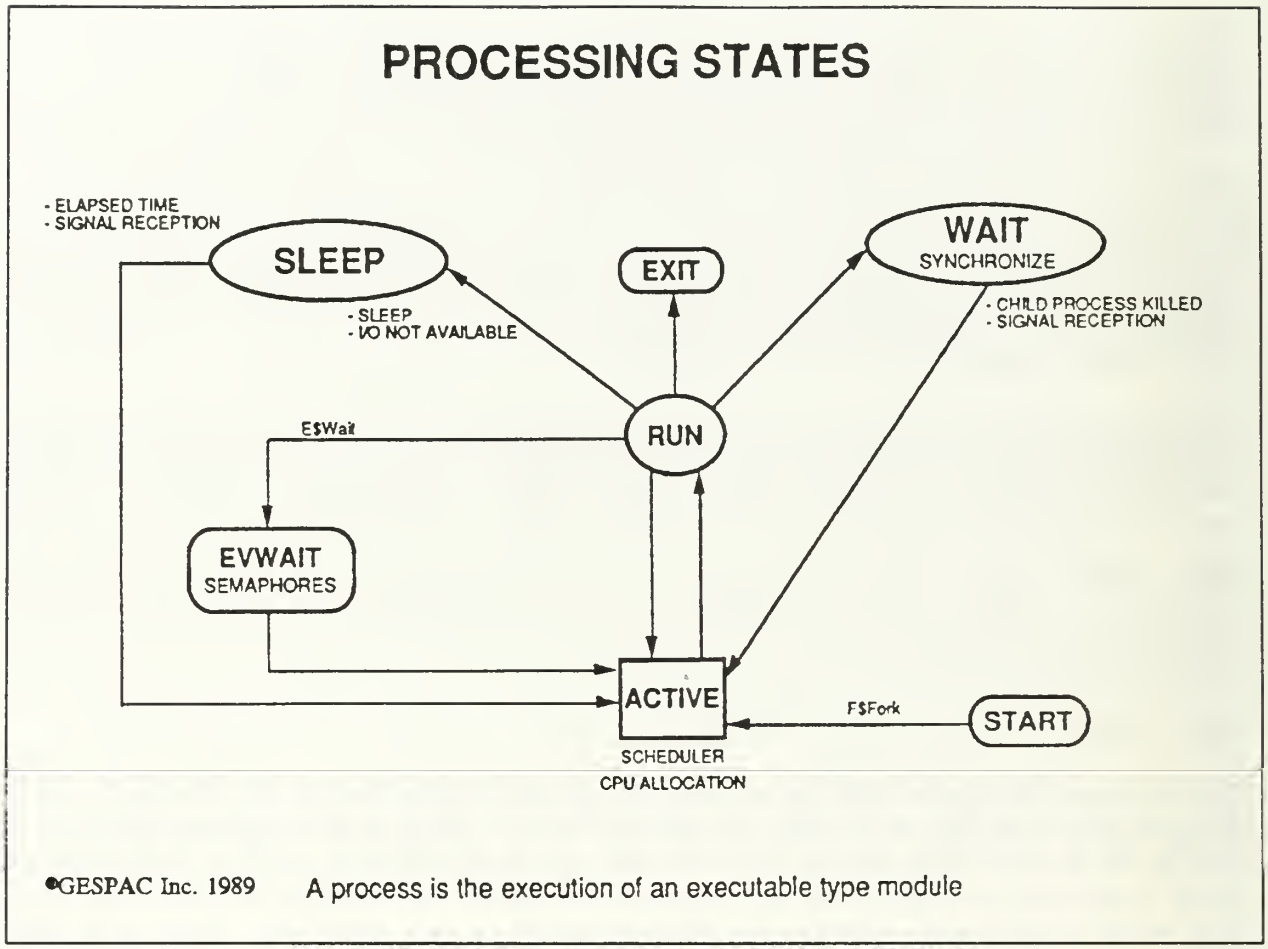

Figure 12 OS-9 Processing States [GES89, pg GTT149]

4.5 CURRENT PROBLEM AREAS AND FUTURE RESEARCH

Deadlock detection in a real-time vehicle should be guaranteed by designing a special periodic real-time process for that purpose. An example can be shown using the NPS AUV software module information in Figure 11 and Table I. The NPS AUV has a tight inner control loop which includes the Mission Executor and Autopilot that must completely repeat on a frequent periodic basis of approximately one second. These two periodic processes might be required to toggle state variables every time the one-second control cycle is successfully completed. Failure to do so after several seconds would be a clear indication of some type of critical problem such as deadlock. Recovery after deadlock detection could be promptly accomplished by reinitializing vehicle control loop software. This new approach to real-time deadlock detection is a straightforward solution to a problem which is frequently considered intractable in non-real-time operating systems.

Deadlock prevention is expensive but essential because the independent, unmonitored and uncontrolled nature of an AUV makes reliability paramount for vehicle survivability. Redundant approaches to deadlock prevention, deadlock detection and deadlock recovery would be prudent. Resolution of deadlock is a particularly sensitive area, given the frequently changing NPS AUV software and the unpredictable ordering of process preemptions and interactions in real time.

As various software modules are integrated into an AUV, software engineering considerations become very important. Key issues are systems integration, verification and validation of process behaviors despite real-time interaction uncertainties, software version control, and system software upward compatibility for integration of future software modules. Failure to methodically address software engineering issues will undoubtedly lead to unpredictable vehicle behavior and tremendous amounts of time wasted troubleshooting individual software modules rather than subtle faults in the operating system implementation.

Fault tolerance is also needed to guarantee overall vehicle reliability and robustness. The approach taken should primarily rely on software checks, rather than the use of redundant processors found in some larger vehicles [Hal91]. Fault tolerance requirements will need to be specified for the top-level mission executor as well as all individual processes. It should also be noted that selection of a distributed multiprocessor architecture allows hardware-based fault tolerance, since failure of a given node could be functionally corrected by reloading and sharing the lost software modules on working processors.

Further work is needed to define formal specifications, characteristics and timing constraints for all NPS AUV software modules. Software module specifications should include inputs and outputs, functionality, module dependencies, hard or soft scheduling constraints, periodic or aperiodic execution, relative priorities, expected frequency and duration, and all other parameters of importance to integrated system design.

The top priority for NPS AUV operating system software integration is to establish a new baseline architecture of system software running in the vehicle. This will allow more sophisticated operations and the addition of new processes to the basic control loop. It is unfortunate that most theses written about the NPS AUV to date have been unable to test their conclusions on the actual vehicle in the water.

Ensuring maximum processor utilization through improvements to the rate-monotonic scheduling algorithm is important work that should continue by verifying current scheduling conclusions using actual NPS AUV processes. The incorporation of dynamic scheduling features holds great promise for the effective coordination of numerous distributed artificial intelligence software modules.

Perhaps the most interesting research immediately applicable to the NPS AUV is the investigation of alternate system software architecture organizations. Many possibilities are available which might incorporate multiple intelligent agents, low-level behaviors, expert systems and blackboard paradigms [WGF88, DL88]. A real-time architecture that allows flexible support of a variety of compatible software approaches should provide the best framework for rapid research progress.

The NPS AUV is a key project that integrates many of the critical technologies important to the Navy of tomorrow. The successful establishment of a reliable and robust real-time system software architecture will be the foundation that supports all future NPS AUV operations.

5 CONCLUSIONS

In this report we have introduced the basics of real-time systems: the relevant terminology, the characteristics of real-time operating systems, and the language requirements and constructs needed for real-time programming to express timing constraints. It is clear that the control of real-time systems is a vital challenge for software engineers, architects, and programmers. This is especially true as more aspects of our daily lives are placed under the responsibility of these systems.

The key to success for a real-time software controller is in the development and execution of a valid schedule. The tasks which constitute the functioning of the real-time system must be identified and then analyzed to determine deadlines. Once these parameters are determined, the real-time system must schedule the tasks in such a manner that deadlines are met regardless of the events presented to the system. Thus, control of real-time systems is essentially a scheduling problem, albeit a sophisticated and complex one.

To meet the challenges dictated by the characteristics of real-time systems, the following issues remain to be resolved:

- Formal methodologies for specification and verification of real-time systems requirements.
- Design techniques which take into consideration the timing requirements of the real-time system from the beginning of the design process.
- Real-time programming languages which have explicit constructs to express timing constraints.
- Sophisticated scheduling algorithms that can handle complex tasks with timing, resource, and precedence constraints in a dynamic and integrated behavior.
- Operating system primitives which manage resources in a real-time system environment in a predictable manner.
- Efficient communication protocols for handling timed messages.
- Sophisticated fault tolerance techniques for time-constrained communication.
- Predictable operating system primitives.

REFERENCES

[Ada90] *Ada 9X Project Report: Ada 9x Requirements.* Office of the Under Secretary of Defense for Acquisition, Washington, D.C., Dec 1990.

[Bar89] J.G.P. Barnes. *Programming in Ada* (3rd Edition). Addison-Wesley Publishing Co, Reading, Mass, 1989.

[Bob91] D.G. Bobrow. "Dimensions of Interaction," *AI Magazine*, Vol 12, No 3, Fall 1991, pp 64-80.

[Boo87] G. Booch. *Software Engineering with Ada* (2nd Edition). The Benjamin/Cummings Publishing Co, Menlo Park, CA, 1987.

[Bru91] D.P. Brutzman. *NPS AUV Integrated Simulator.* Masters Thesis, Naval Postgraduate School, Monterey, CA, Mar 1992.

[BS86] T.P. Baker and G.M. Scallon. "An Architecture for Real-Time Software Systems," in [SR88]; reprint from *IEEE Software*, May 1986, pp 50-58.

[BW90] A. Burns and A. Wellings. *Real-Time Systems and Their Programming Languages.* Addison-Wesley Publishing Co, Reading, Mass, 1990.

[Cra88] B. Cramer. "Writing Real-Time Programs under UNIX," *Dr. Dobb's Journal*, Vol 13, No 6, Jun 1988, pp 18-29.

[Dei90] H.M. Deitel. *An Introduction to Operating Systems* (2nd Edition). Addison-Wesley Publishing Co, Reading, Mass, 1990.

[Dib88] P. Dibble. *OS-9 Insights: An Advanced Programmers Guide to OS-9/68000.* Microware Systems Corp, Des Moines, Iowa, 1988.

[DL88] E.H. Durfee and V.R. Lesser. "Planning to Meet Deadlines in a Blackboard-based Problem Solver," in [SR88], pp 595-608.

[DLA91] P. Dasgupta, R.J. LeBlanc, Jr., M. Ahamad, and U. Ramachandran. "The Clouds Distributed Operating System," *Computer*, Vol 24, No 11, Nov 1991, pp 34-44.

[DOD83] US Department of Defense. *Reference Manual for the Ada Programming Language.* ANSI/MIL-STD 1815, 1983.

[Fal88] H. Falk. "Developers Target UNIX and Ada with Real-Time Kernels," *Computer Design*, Vol 27, No 7, 1 Apr 1988, pp 55-70.

[FM91] I.M. Flynn and A.M. McHoes. *Understanding Operating Systems.* Brooks/Cole Publishing Co, Pacific Grove, CA, 1991.

[GES89] GESPAC Inc. *Introduction to OS-9/68000.* GESPAC Inc, Mesa, AZ, 1989.

[Hal91] P. Hale. DARPA UUV Project Manager, C.S. Draper Laboratories, Cambridge, Mass. Interview by D.P. Brutzman, 11 Dec 1991.

[Hil88] D. Hildebrand. "Message-Passing Operating Systems," *Dr. Dobb's Journal*, Vol 13, No 6, Jun 1988, pp 34-48.

[HMC90] A.J. Healey, R.B. McGhee, R. Cristi, F.A. Papoulias, S.H. Kwak, Y. Kanayama, Y. and Lee. "Mission Planning, Execution, and Data Analysis for the NPS AUV II Autonomous Underwater Vehicle," *Proceedings of the First IARP Workshop on Mobile Robots for Subsea Environments*, Monterey, CA, Oct 1990, pp 177-186.

[Kah91] D.K. Kahaner. "TRON (The Real-Time Operating System Nucleus)," *Scientific Information Bulletin*, Vol 16, No 3, Office of Naval Research Asian Office, Jul-Sep 1991, pp 11-19.

[Kas88] H. Kasahara. "Parallel Processing of Robot-Arm Control Computation on a Multimicroprocessor System," in [SR88], pp 576-585.

[KL91] K.B. Kenny and K-J. Lin. "Building Flexible Real-Time Systems using the Flex Language," *Computer*, Vol 24, No 5, May 1991, pp 70-78.

[KS86] E. Klingerman and A.D. Stoyenko. "Real-Time Euclid: A Language for Reliable Real-Time Systems," in [SR88]; reprint from *IEEE Transactions on Software Engineering*, Sep 1986, pp 941-949.

[Lea91] B. Leatherman. *An Approach to Integration of Real-Time Software for Autonomous Underwater Vehicles*. Masters Thesis, Naval Postgraduate School, Monterey, CA, Sep 1991.

[LG85] I. Lee and V. Gehlot. "Language Constructs for Distributed Real-Time Programming," *Proceedings of the Real-Time Systems Symposium*, San Diego, CA, Dec 1985, pp 57-66.

[Mak91] D. Makris. *Real-Time Scheduling and Synchronization for the Naval Postgraduate School Autonomous Underwater Vehicle*. Masters Thesis, Naval Postgraduate School, Monterey, CA, Mar 1992.

[NL88] S. Natarajan and K. Lin. "FLEX: Towards Flexible Real-Time programs," *1988 IEEE International Conference on Computer Languages*, Oct 1988, pp 272-279.

[Nel91] M.L. Nelson. *CS4112 (Operating Systems) Course Notes*. Naval Postgraduate School, Monterey, CA, Fall, 1991.

[PM87] D. Pountain and D. May. *A Tutorial Introduction to OCCAM Programming*. BSP Professional Books/ Blackwell Scientific Publications Ltd, Palo Alto, CA, 1987.

[SR88] J.A. Stankovic and K. Ramamritham. *Tutorial: Hard Real-Time Systems*. Computer Society Press of the IEEE, Washington, D.C., 1988.

[TS89] C. Tomlinson and M. Scheevel. "Concurrent Object-Oriented Programming Languages," in *Object-Oriented Concepts, Databases, and Applications*, edited by W. Kim and F.H. Lochovsky. Addison-Wesley Publishing Co/ ACM Press, New York, NY, 1989.

[WW85] W. Werum and H. Windaver. "Introduction to PEARL Process and Experiment Automation Realtime Language," Friedr. Vieweg & Sohn, 1985.

[WGF88] M.L. Wright, M.W. Green, G. Fiegi, and P.F. Cross. "An Expert System for Real-Time Control," in [SR88], pp 586-594.

DISTRIBUTION LIST

Center for Naval Analyses
4401 Ford Avenue
Alexandria, VA 22302-0268 — 1 copy

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314 — 2 copies

Egyptian Military Attache
2308 Tracy Place NW
Washington, DC 20008 — 2 copies

Egyptian Armament Authority - Training Department
c/o American Embassy (Cairo, Egypt)
Office of Military Cooperation
Box 29 (TNG)
FPO, NY 09527-0051 — 1 copy

Military Technical College (Egypt)
c/o American Embassy (Cairo, Egypt)
Office of Military Cooperation
Box 29 (TNG)
FPO, NY 09527-0051 — 5 copies

Military Research Center (Egypt)
c/o American Embassy (Cairo, Egypt)
Office of Military Cooperation
Box 29 (TNG)
FPO, NY 09527-0051 — 1 copy

CAPT A. Beam, USN
DARPA Undersea Warfare Office
PRC Inc.
1550 Wilson Blvd, Suite 600
Arlington, VA 22209 — 1 copy

Director of Research Administration, Code 81
Naval Postgraduate School
Monterey, CA 93943 — 1 copy

Library, Code 0142
Naval Postgraduate School
Monterey, CA 93943 — 2 copies

Dr. G. Bradley
Operations Research Dept., Code ORBz
Naval Postgraduate School
Monterey, CA 93943 — 1 copy

LTCOL S.M. Badr, Egyptian Army
Dept. of Computer Science, Code CS
Naval Postgraduate School
Monterey, CA 93943 — 5 copies

LCDR D.P. Brutzman, USN
Dept. of Computer Science, Code CS
Naval Postgraduate School
Monterey, CA 93943 — 2 copies

MAJ R.B. Byrnes, USA
Dept. of Computer Science, Code CS
Naval Postgraduate School
Monterey, CA 93943 — 2 copies

Dr. R.B. McGhee
Dept. of Computer Science, Code CSMz
Naval Postgraduate School
Monterey, CA 93943 — 1 copy

Dr. M.L. Nelson, MAJ, USAF
Dept. of Computer Science, Code CSNe
Naval Postgraduate School
Monterey, CA 93943 — 5 copies

Mr. R.H. Whalen
Dept. of Computer Science, Code CS
Naval Postgraduate School
Monterey, CA 93943 — 1 copy